Dynamic, Scalable and Flexible Resource Discovery for Large-Dimension Many-Core Systems

Javad Zarrin, Rui L. Aguiar, João Paulo Barraca

Javad Zarrin {javad@av.it.pt} Instituto de Telecomunicações - Aveiro. Rui L. Aguiar {ruilaa@ua.pt} Universidade de Aveiro, Portugal. João Paulo Barraca {jpbarraca@ua.pt} Universidade de Aveiro, Portugal.

Abstract

Future large scale systems will execute novel operating systems running across many chips with many cores. In this highly distributed environment, resource discovery is an important building block. Resource discovery aims to match the application's demands to the existing (distributed) resources, by discovering and f nding resources at run-time, and then selecting the best resource that matches the application running requirements. The main contribution of this paper is the design and evolution of a highly scalable, highly f exible, resource discovery model for such heterogeneous environments. The model is based on self-organizing processing resources in the system according to a hierarchical resource description where each group of resources has a local directory that collects and keeps the information of the underlying resource members (cores) in dif erent layers. Operationally, at each layer, it consists of a peer-to-peer architecture of modules that, by interacting with each other, provide a global view of the resource availability in a large, dynamic and heterogeneous distributed environment. The proposed resource discovery model provides the adaptability and f exibility to perform complex querying by supporting a large set of signif cant querying features (such as multi-dimensional, range and aggregate querying) while supporting exact and partial matching, both for static and dynamic object contents. The paper demonstrates by simulation how the proposed model can deal with issues such as scalability, ef ciency and adaptability of resource discovery in future many-core systems which are the major challenges in the current state of the art. The simulation show that the proposed resource discovery model can be applied to arbitrary scales of dynamicity, both in terms of complexity and of scale, positioning this proposal as a good architecture for future many-core systems.

Keywords: many-core systems, distributed operating system, service oriented operating system, resource discovery, resource management, resource sharing, DHT, P2P, grid and cloud computing, cloud-like systems, distributed systems, resource description, overlay networks

1. Introduction

Processors will not be able to enhance their (single-thread) performance exponentially[1]. Rather, due to the many-core nature of future large scale computing platforms, the evolution will proceed by scaling the number of processing cores. Consequently, application software will no longer necessarily get faster execution speeds automatically with each hardware upgrade, but will have to be adapted to get exposed to the higher level of parallelism by the CPU. This means that to benef t from the many-core hardware improvements, we should reconsider our traditional concepts of applications, operating systems and compilers towards the direction of massively distributed, heterogeneous and dynamic computing environments. We can imagine that computing nodes in future High Performance Computing (HPC) systems, with thousands of cores, may be connected together to form a single transparent computing unit, thus hiding the complexity and distributed nature of the many-core system from applications while it is expected that the computational systems extend far beyond the chip, with hundreds and thousands of chips incorporated into collaborative execution units in a wide-scale distributed structure.

Moreover, due to the specif c requirements and limitations of the current HPC systems, particularly in terms of high dynamicity and high heterogeneity (e.g. the static conf guration of the task execution environment which usually depends on specif c applications, libraries, job schedulers and operating systems, restricts the service users to implement certain application scenarios) and also to the steady progress of Service-oriented and Cloud computing paradigms, we can envision that the long-term future of HPC clusters will tend towards large scale distributed cloud-like systems supporting high performance computing with increasing f exibility and ef ciency.

For distributed operating systems executed on such Cloud-like many-core-enabled computing environments, resource discovery (RD) is a vital building block to maximally exploit the capabilities of all distributed heterogeneous resources. In fact, when we have a pool of variable-type and large number of processors, resource sharing becomes complex. This is specially true if we are trying for overloaded processors to potentially migrate some applications to other (possibly dif erent) processors in the system. But before resource (re)allocation and execution migration, we need to f nd resources and locate them.

Resource discovery challenges for the next generations of many-core systems can be considered mainly as associated to

scalability and efficiency issues. The description of arbitrary resources for a huge number of heterogeneous resources in an adequate and efficient manner is not practically attainable at present. All the diverse resources in a distributed system need to be defined by a set of strict parameters that adequately describe the characteristics and performance factors of the correspondent resources sufficiently. However, exploring all the existing cores, when different architectures and features are bound, and discovering the ones apt for a certain set of conditions on the local chip or on a larger scale network is almost unfeasible due to potential excessive information exchange. Therefore discovery mechanisms should be efficient enough to support large scale distributed heterogeneous (many-cores, and not only multi-cores) environments. Furthermore, with the potential high dynamicity of resources, providing an intelligent real-time resource awareness mechanism for managing and scheduling resources (cores), which is able to estimate an optimum resource allocation for a specified request, is a major challenge.

We must note that values such as clock rate, MIPS, GFLOPS, cache size, etc., are useful metrics to describe computing units, but they are not reliable enough and applicable to operate as complete resource descriptions[2]. Micro Benchmarks such as HPL, NAMD, or SPEC[3] attempted to solve this problem from the application side, testing a set of standard algorithms on the target system. However, there is still not an entirely accurate and precise performance metric to properly describe resources in a many-core system in general. The problem of resource identification and discovery, respectively can be structured along the following two questions:

1. How to identify the resources required for a given process? or How to identify the hardware resources requirements from either the static (source code) or the dynamic (runtime) behavior of the program (i.e., Resource Identification)?
2. How to find the required resources for a given query (i.e., Resource Discovery)?

For optimally running a given process, we need to identify the resources which maximize the matching between resource capabilities and process characteristics. This could be done using a code analyzer component (based on methods such as meta data provided by developer) or examining the characteristics and behaviors which are exhibited by the code itself. For example, in the case of a process with several data inputs and single instruction, perhaps the best matching resource could be a SIMD vector processor. In other example, for a given parallel application containing several threads, according to its dependency graph, the required resources could be a set of processing cores which are connected to each other with the links that satisfy the specific communication requirements.

Upon identification of the adequate resources for a given process, the user or any responsible OS component can generate a query for discovery. Afterwards, the resource discovery module would be responsible to find the best matching resources for the query through efficient exploration of the resources in the whole system.

In this paper, developed under the framework of the S[o]OS project[4, 5], we focus on the second question and present a resource discovery model for distributed operating systems which deals with the aforementioned challenges. S[o]OS (Service-oriented Operating System) was a European project aiming to generate a reference architecture for future large scale, distributed infrastructures, considering the current trends in processor manufacturing including many-core systems, the memory wall problem, the growing degree of heterogeneity, high-dynamicty of the resources and the different hierarchies within the communication infrastructure.

The main contribution of this paper is to design and implement a new resource discovery (RD) model for the large scale distributed cloud-like systems with the following general preconditions and assumptions:

1. The heterogeneity of the resources is very high. i.e., high-heterogeneity in terms of computation and communication characteristics such as heterogeneity of the processing resources (e.g. CPU, GPU, etc.), interconnecting and communication networks, memory hierarchies, etc.
2. The environment is highly dynamic, so that no static configuration would be possible.
3. The execution "threads" are on the level of tasks or services.

Due to space restrictions, and for a better illustration and evaluation of our solution, in this work we discuss mostly RD from the point of view of computational capacity, expressed by the capabilities of the Central Processing Unit (CPU). Nevertheless, our RD method is fully general, applicable to all resources needed in a large future computational systems with suitably adapted discussion.

The rest of paper is organized as follows: In section II we describe our proposed resource discovery model for large dimension many-core systems. In section III we present the evaluation results. Section IV describes related works and finally section V presents our conclusion.

2. Proposed Resource Discovery in Many-Core Systems

Current RD solutions [6] are either directory based or fully distributed approaches, trading reliability and efficiency by delay and overhead. Our proposed RD method tries to provide a bridge between these two approaches. Our protocol makes use of a hierarchical structure, albeit relying on chord based [7] [6], distributed hash tables (DHTs) to maintain resource information. By resources, we consider all the entities that compose a computational system, such as its Random Access Memory (RAM) modules, CPU, Network Interface Controller (NIC), interconnects (or network links), special processing and Graphic cards, as well as their operational status and performance metrics. We endeavour to make this information potentially available to all systems in a heterogeneous, many-core environment so that the processes are distributed in the most effective and efficient manner. The decisions can then be taken at each core, in a truly distributed fashion.

We consider DHT as an essential technology that allows the design of robust distributed components to provide storage and

lookup services. Despite the existing master/slave data replication architectures often used today, DHTs are more reliable and able to provide performance guarantees [8] in fully distributed systems. In multi-core environments we attempt to avoid the exchange of unnecessary resource discovery information between nodes, to keep overhead to a minimum. To achieve this purpose, we noticed that basic DHT approaches are not enough, and we decided to organize resources in multiple layers, implemented by using hierarchical Chord based [7] Distributed Hash Tables, essentially with resource aggregation and summarization.

2.1. Description of Resources

Figure 1 illustrates the hierarchy model of our resource description system. We categorize all relevant resource information, including computational and communication properties in different layers, going from more abstract information to more detailed characteristics. By using a hierarchical resource description model we can, at each level, just transmit the information required for the specific needs of a given discovery procedure. The model needs to rely on techniques for information aggregation and summarization. Information available at different domains will contain different levels of detail. Aggregation and summarization allow nodes to have coarse views about their neighbors, and then, after deciding that a neighbor is a potential candidate, get detailed information about its resources, if needed. For our model we consider the existence of the following layers:

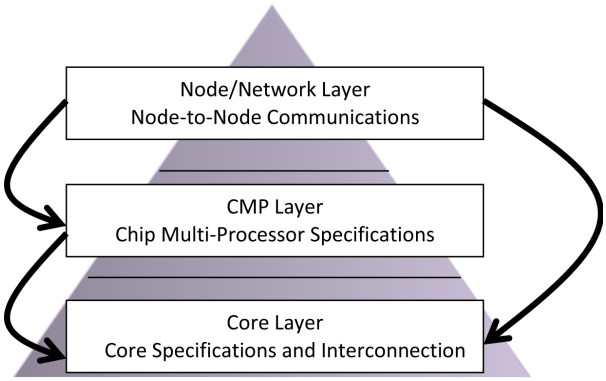

Figure 1: Information architecture for hierarchical resource description

- Layer1 (Inter-Core Level or Network On Chip (NOC) Level): Resource description on this level is characterized by information directly related to core internal operation. The most typical information in this level is related to the description of the cores, performance counters, private cache size or routing paths established inside the NoC. This information is of interest for optimizing the operation inside the chip and managing cores locally.

- Layer2 (Inter-Chip Level): This describes the specifications of the edges for the core communications, interconnection networks and memory hierarchy which include information such as bandwidth, latency, shared and/or global cache size, as well as the number of cores available and their characteristics. This information is relevant to the management of several boards in the same host. Traditional operating systems take into account this information when scheduling tasks in Symmetric Multi Processing (SMP) [9] systems.

- Layer3 (Inter-Board Level): Resource description here covers a total overview of all the communications, memory and processing features and aspects in the nodes, such as memory size and number of processors, and coarse performance counters, as well as inter-node communication information and its network structure. This aggregate information is available to other nodes to optimize task placement across hosts in the same operational domain (see Table1).

The distribution of resource information in the system is schematically depicted in Figure 2. According to this scheme, each node has the possibility of getting directly the highest level of information about itself and its close neighbors in different circles of vicinity. Furthermore, each node has some summarized information about remote nodes, enough to provide an overall perspective of the whole system from its point of view. It is obvious that nearby resources with lower latency will be preferred for resource selection in the discovery procedure. This is particularly relevant if the tasks which are to be distributed have input and/or output dependencies associated with the local devices. Task placement must take in consideration both processing capabilities and communication (I/O, network) latency.

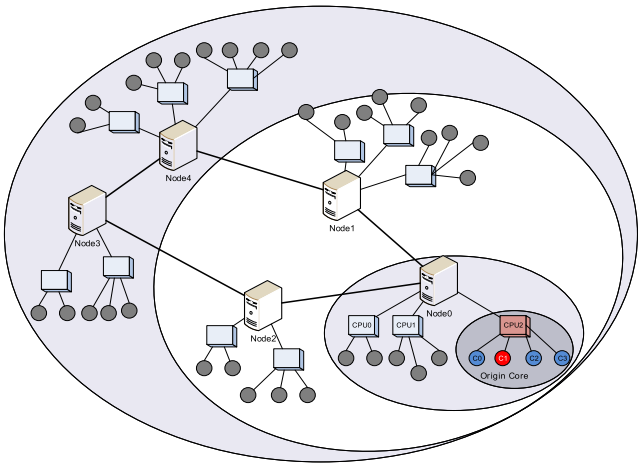

Figure 2: Fish eyes's model of resource description for resource discovery

We describe resources by using a set of attributes which are structured in two different classes: individual and collective attributes. Individual attributes are the particular properties and characteristics for each resource. Collective attributes describe the communication properties between individual members inside a group of resources or group of resource groups. Link properties are necessary to create a descriptive graph of the interconnected resources which can be employed for mapping between application segments and required set of resources. Attributes are static (such as Cache Size, Frequency, Processor Type, number of cores, etc.) or dynamic (such as CPU

Table 1: Examples of resource attributes in each layers

Layers	Communication	Attributes
Layer3	Node2Node	Core Clock Rate, L1 Cache Size, L1 Latency, L2 Cache Size, L2 Latency, L1 Cache Line, L2 Cache Line, L1 Cache Associativity, L2 Cache Associativity, MIPS, Number of ALUs, Type of ALUs, ALU Functionalities, Number of Data-PU Channels, Number of Instruction-PU Channels, Vector Length, L3 Latency, Memory Latency, etc.
Layer2	Die2Die	BUS Frequency, Memory Bandwidth, Memory Frequency, Cache Coherence, ISA, Micro Architecture, Interconnection Network, Size of Processor Address Bus, Size of Processor Data Bus, Processor Name, Processor Class, Processor Type, Number of Cores, etc.
Layer1	Core2Core	Memory Size, Number of Dies, Network Bandwidth, Network Latency, Total Number of Cores, etc.

load, available memory, available bandwidth, etc.). The static attributes are not required to be updated in short intervals, however dynamic attributes are very sensitive and they might change very frequently.

2.2. Node Types

We store the resource information discussed above in different layers and in a ring-based distributed hash tables where resources are placed in a ring according to their hashed keys. With these hashing functions we can map all attribute values in a specific layer to the same m-bit space in a DHT ring. We must consider that depending on the resource discovery approaches, it is not necessary to use DHT in all layers. Specifically for the upper layers we can store resource information in the format of the layer stamps which can be validated according to a set of conditions in the incoming query templates. In our system we use different procedures at different layers by efficiency/complicity trade-offs.

We define our system architecture according to three different types of nodes: super-nodes, aggregate-nodes and leaf-nodes. (For the discussion in this paper, we assume that a node is supported by a physical core.) Each of these node types take position in a layer within the hierarchy.

Leaf-nodes are in Layer1, so all the nodes in this layer maintain their own resource information plus a finger table which handles forwarding lookup request to other nodes in its DHT ring. Leaf-nodes have the ability to run the resource discovery procedure which generates and sends a resource request query that includes a m bits key to an aggregate-node that hosts a Query Management Service(QMS).

Leaf-nodes communicate with their aggregate-nodes and the aggregate-nodes establish a structured DHT-based overlay in the form of a Chord ring among their leaf-nodes. For instance, all the cores in a CPU can be considered as leaf-nodes, bearing in mind that there is at-least one core per CPU which runs a service that behaves as the aggregate-node, a central contact point that summarizes all the information in that CPU.

Aggregate-nodes address the information on layer2. Query Management Service(QMS) instances are the information service modules that reside in aggregate-nodes. QMS gets a query from a leaf-node and then starts the lookup procedure to find the required reply. If the reply is found in the local QMS domain, it will be sent back to the requester. Otherwise the QMS forwards the query to other aggregate-nodes in its neighborhood. It will use a probability based method to select the next

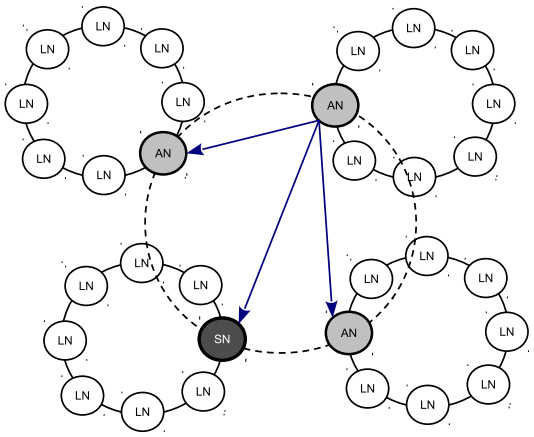

Figure 3: Types of node in our hierarchical structure

aggregate-nodes among others. In fact, each aggregate-node in our many-core system is a QMS host and in this paper we use "QMS" to refer to the aggregate-node.

Super-nodes are in Layer3. If the resource discovery can't find a specified resource after searching all the QMS in the local board, the query message will be forwarded to other neighbor nodes in the network by a SQMS (super-node QMS) which is hosted on the super-node. This Super-node has information about all the adjacent super-nodes in the network. Accordingly, it uses Any-cast mechanism to select the subsequent super-node among others with the minimum latency (see Figure 3 and 4).

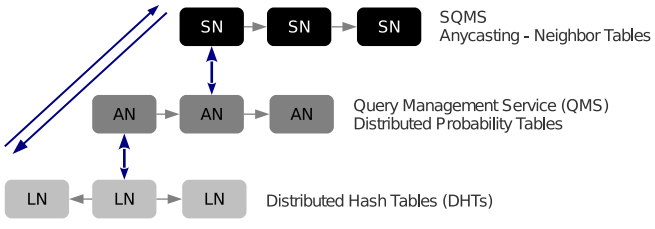

Figure 4: Transaction of the query messages between layers

2.3. Discovery Processes and Algorithms

According to the characteristics of each resource, we can extract the values for our set of attributes. Conforming to the

resource requester's priority list of the attributes, multiple attributes' values of each processor are extracted and encoded in order to form an m-bit key which is used to update and locate that resource. In the system with $d1$, $d2$ and $d3$ attribute dimensions, in each layer, we need to generate a set of keys ($k1$,$k2$ and $k3$) with $m1$, $m2$ and $m3$ bits for all the layers. A query message consists of three parts ($c1$, $c2$, $c3$), each representing a set of conditions for satisfactory resources in each layer. The below script is the general format of the resource discovery query messages.

$Query : Condition1 \wedge Condition2 \wedge Condition3 \wedge ...$
$Condition : AttributeOperatorValue \wedge AttributeOperatorValue \wedge ...$
$Operator : (<, \leq, =, \geq, >)$

In the DHT ring we ensure that any leaf-node is able to discover other resources by using the resource m-bit key up to a certain limit of time with the order of log N, where N is the number of leaf-nodes in Layer1. In the inter-core layer, discovery proceeds in a multi-hop fashion with each of the leaf-nodes maintaining its small finger table containing the information about other leaf-nodes in the group. They forward the discovery message recursively to the leaf-node that is closest to the $m1$-bit key of the original requester's query.

In order to implement layers hierarchically, Hierarchical-DHT is the most efficient logical design[10], which best corresponds to the underlying physical hierarchical structure of the core-chip-node in the many-core system which stores key-data pairs by assigning keys to different processing units (cores) according to their positions.

The Query Management Service (QMS) components, which provide the distributed directory service, employ H-Chord DHTs to keep resource description available to all the cores in a chip. Each QMS maintains the description values for all the keys for which it is responsible, and for each core there is only one unique key. Chord specifies how keys are mapped to cores, and how a node can find data for a specific key or range of keys by first locating the nodes which are responsible for that particular key. We can summarize the QMS responsibilities and functionality as follows:

1. Start-up, building DHT overlays and gathering information.
2. Entry point for the ring Lookup, requesters send their resource queries to their local QMS.
3. Maintaining resource cost table, retaining information about other QMSs which reside on the other aggregate-nodes in neighborhood such as neighbor ID, probe factor and the Inter-Chip layer information. Probe factor is the probability to find a specific requested resource in a remote QMS domain.
4. Retaining the local Inter-Chip layer information, a summarized set of characteristics shared among all the leaf-nodes in a group.
5. Sustaining information about the local super-node which is the representative of several leaf-nodes groups in the system.
6. Processing, managing and forwarding of the query message between layers.

At layer3, the super-nodes are able to send the query to a group of super-nodes in vicinity that are using the same Any-cast address. The actual number of receivers can range from only one to several group members, where some of the receiver nodes satisfy the query requirements for metrics such as latency, availability and load balancing between the target resources. Algorithm 1 below describes the process of resource discovery in a large cluster with multi-dye and multi-core nodes. Before discussing the steps of this algorithm we need to explain the following terms (concepts):

- Discovery Event: It is an event of the discovery procedure which is assigned to the processing resources in the network (which are supposed to operate as Resource Requester), triggered when the corresponding processing resource has execution overload, and later proceeds distributing the overloaded processing tasks over other available resources. To put it briefly it is the trigger to initialize the procedure of discovery for the required resources.

- QMS: Query Management Services are the components which reside in each CPU. They are responsible for checking the availability of their processing units and managing the individual queries from various origin cores.

- RCT: Resource Cost Table is a data structure located in each aggregate-node to record the information about the other neighboring aggregate-nodes. This information consists of the probability factors for each neighbor which is based on the cost of remote resources. After searching the local resources, the local QMS must forward the lookup request to other remote QMSs which are adjacent to the current one. The decision to make a selection of the next QMS is done according to the value of the probe factors in RCT. (see Algorithm 2 later in this section)

The procedure of resource discovery takes place according to the following steps(see Algorithm 1):

1. System initialization and resource description: In this step the many-core system is completely described, and components such as QMS, and Discovery Event are assigned to the processing units.
2. Once a core determined an overload execution, it triggers a discovery event to distribute extra load over to other cores in the system.
3. Discovery event generates a query which requests on-demand resources from local and remote nodes. The query is generated according to the application requirements and parameters. For instance, depending on the type of application, queries can be produced to find the most efficient resources due to Flynn's classifications [11]. Queries consist of three types of conditions for each layer and they would be represented in a format like $c1.c2.c3$. If the requester doesn't prioritize highly its required conditions in a particular layer, the resource description checking on that layer will be ignored. For example a query like $c1.N.N$ means that we do not need to investigate on layer 2 and 3, but we just need to explore the system to find a match on layer1(see Algorithm 1).

Algorithm 1: Resource Discovery General Procedure

Input: applicationArguments ; /* description of the required resources */
Output: discoveredResources ; /* list of discovered resources */
Data: RCT ; /* resource cost table */
Data: reqMsg, repMsg, QMS, remoteQMS ; /* query management service */
Event: discoveryEvent; discoveryEvent.setTrigger;
Collection: manyCoreSystem:=Set{n, $f(i)$} ; /* n nodes, $f(i)$ cores per $node_i$ */
Collection: multiCastGroup, qRes(qualified recurses), dRes(discovered resources);
foreach *node* ∈ *manyCoreSystem* **do**
    node.nodeAssign(QMS);
    foreach *core* ∈ *node.requesters* **do**
        core.coreAssign(discoveryEvent, RCT);
On-discoveryEvent:
$query_{main}$=generateQuery(applicationArguments);
core.set(original-requester);
QMS=getLocalQMS($core_{OR}$);
reqMsg=generateReqMsg($query_{main}$,$core_{OR}$);
send(reqMsg,QMS) ; /* send query from the source core (original requester) */
On-receive(QMS):
subQueries=queryAnalyzer.divide(reqMsg);
foreach *query* ∈ *subQueries* **do**
    forward query to the proper *layer* if it is required;
if $QMS.layer_{proprties}$ *satisfies the query* **then**
    results=QMS.Lookup(*query*);
    if *query is fully resolved or query is expired* **then**
        send(results,QMS_{OR});
    else if *query is partially resolved* **then**
        send(results,QMS_{OR});
        forward query to the proper *layer* if needed;
        remoteQMS=RCT.selectNextQMS();
        send(updated *query*,*remoteQMS*);
    else
        forward query to the proper *layer* if needed;
        remoteQMS=RCT.selectNextQMS();
        send(*query*,*remoteQMS*);
else
    forward query to the proper *layer* if needed;
    remoteQMS=RCT.selectNextQMS();
    send(*query*,*remoteQMS*);
On-receive(SQMS):
forward query to the proper *layer* if needed;
multiCastGroup=SQMS.getNeighborGroup();
Broadcast(*quey*,multiCastGroup, Any-cast);
On-receive-queries-results(QMS_{OR}):
qRes.add($query_{main}$, results);
if *discovery is completed or main query is expired* **then**
    dRes=queryAnalyzer.converge($query_{main}$, qRes);
    repMsg=generateRepMsg($query_{main}$, dRes);
    send(repMsg,$core_{OR}$);

4. Origin core (resource requester, or the one that has overload) is a processor that starts query dissemination. It sends a message(containing the main-query) to its local QMS and checks the surrounding processors. In fact, when a requester sends a main-query to a QMS node, the QMS would analyze that request and in function of the main-query requirements, the QMS propagates one-to-several sub-queries around the system with similar TTL. Due to the type of each query (i.e., sub-query), the local QMS decides to direct them to a proper layer using one of the below rules:
   Action 1- $c1.N.N$: Starts a local DHT lookup to find a resource that satisfies the $c1$ conditions.
   Action 2- $c1.c2.N$: Checks the layer2 information of the local QMS and if it fits $c2$ conditions it continues with action one, else the local QMS checks RCT to select the subsequent matching remote QMS which manages a group of leaf-nodes.
   Action 3- $c1.c2.c3$: Forwards the query to a super-node in higher layer. Once the query reaches a super-node, a QMS that is running on it (which is called SQMS), checks the layer3 information and if it fits $c3$ conditions, it continues with the action two, or it performs a top-level lookup service which routes the query through the network to the network node that is acceptable for the $c3$. During this phase, the query only passes through super-nodes, hopping from one group to the next. A super-node in one group uses its knowledge of the IP addresses of super-nodes in the subsequent group along the route to forward the query message from a network node to another network node. Any-cast is used to decide and select the subsequent super-node.
5. All the receiver processors check the query messages, their conditions and requirements, and according to the resource description, they will be added to a collection of the qualified resources if they can fulfil the query requirements.
6. If all the potential local processors fail to satisfy a specified query and the collection of qualified resources gets empty, the discovery procedure checks the list of remote SQMS and sends Any-cast queries to all the remote resources neighboring the Origin group (i.e., a group of processors with the same SQMS address where the original requester belonged). This process will continue within increasingly larger neighborhoods until one of these two situations happens; (a) the query is fully resolved. (b) the query timeout is expired. In both of the aforementioned situations, the QMS_{OR} (the QMS of the original requester) would be updated with the query results. In other situation, when a query is partially resolved, an update message containing the query partial results will be transferred to QMS_{OR} while a new version of the query continues the search within the current layer or the upper layer in order to find the rest of requested resources.
7. Finally after finding qualified resources for a particular query (i.e., sub-query), the query reply follows the reverse path already taken by the query message to QMS_{OR} through the overlay network. It updates the probe factor values in intermediate nodes and RCTs for their next us-

age. The main-query which has already registered in the QMS_{OR} would be completed either if all the sub-queries are resolved or if the main-query is expired. Upon completion of the main-query, the optimum results would be converged into the collection of discovered resources which would be transmitted to the original requester by a reply-message. The optimum results would be calculated by considering also the required communication capacities among the qualified resources. We must note that, on the time of main-query expiration, some of the sub-queries may be resolved while others may not have returned any results. In such a case the QMS of the original requester will send the reply message corresponded to the main-query to the original requester containing the optimal partial gathered results.

Algorithm 2: Query Forwarding in $Layer_2$

```
InputQuery= Query(c1,c2,c3) ; /* c1,c2,c3: set of
query conditions in layer1,layer2,layer3 */
Collection: L1,L2,L3;
foreach record in RCT do
    if record.k2 is a valid key according to c2 then
        add record to L1;
    else
        add record to L2;
;       /* k2: representation of the resource
characteristics in layer2 as a key */
L1.sort(probe factor); L2.sort(probe factor);
L3=L1+L2;
forward(InputQuery,L3[0].nid);
```

2.4. Multi-Dimensional Discovery

The description of resources is not only supposed to support single attribute based discovery, rather it should be able to be used for multi-dimensional range queries. Therefore for multi-dimensional abstraction in a system in which the information of resources are distributed in hierarchical layers, we assume our abstraction system to have L layers $l_1, l_2, .., l_L$ and each resource to have A attributes $a_1, a_2, .., a_A$. Then according to the layers description each attribute a_i is placed in the layer l_j:

$a_i \in l_j$ *where* $1 \leq i \leq A$ *and* $1 \leq j \leq L$

Each attribute is described by

$< At\_ID : Op\_ID : At\_VA : La\_ID : At\_TY >$

which its parameters are self describing Attribute ID (At_ID), Operator ID (Op_ID), Attribute Value (At_VA), Layer ID (La_ID) and Attribute Type (At_TY: String or Number).

La_ID can be extracted from At_ID and also we assume bitmap and numeric values to represent the string types so that an attribute can be defined in a triple format like $< At\_ID : Op\_ID : At\_VA >$, where At_ID and At_VA are numerical values and $OP = \{<, \leq, =, >, \geq\}$. Using a hash function we create $H(At\_VA)$ for every At_VA values, besides, in order to support range querying the hash function has to support locality preserving. It means that for each attribute a_i:

$if\ At_{VA}(a_i) \in [VA_{min}, VA_{max}]$

$\Rightarrow H(At_{VA}(a_i)) \in [H(VA_{min}), H(VA_{max})]$

Our proposed resource discovery solution supports four different approaches for range-based multi-dimensional discovery.

Approach A: In the first approach, each resource has to generate hash values for its information in all the dimensions. Afterwards it uses a priority function to place hashed values in an ordered list of PL. In the next step each resource will create a uniform key of H_T which demonstrates all of the resource properties in a specific layer. So for each resource r_i we have:

$H_T(r_i) = f(H(At_{VA}(a_1)), H(At_{VA}(a_2)), .., H(At_{VA}(a_A)), PL)$

Finally each resource r_i has to register its uniform key in $S(r_i)$, where $S(r_i) = successor(H_T(r_i))$.

Approach B: In the second approach we store resource information for each dimension in an individual DHT ring. Therefore each resource r_i with A attributes collaborates in D particular DHTs where $A = D$. For example in order to perform a range query in such an environment, we will be looking for resources with set of specific attributes $a_1, a_2, .., a_A$ in the layer of l_j which attribute values $At_{VA}(a_1), At_{VA}(a_2), .., At_{VA}(a_A)$ should be in following ranges respectively:

$[VA(a_1)_{min}, VA(a_1)_{max}], [VA(a_2)_{min}, VA(a_2)_{max}], ..,$
$[VA(a_A)_{min}, VA(a_A)_{max}]$

We distribute the resource information within layer l_j to D distributed hash tables $H_1, H_2, .., H_D$ where each dimension is hasstored in a DHT:

$H_1(At_{VA}(a_1)), H_2(At_{VA}(a_2)), .., H_D(At_{VA}(a_A))$

To resolve a query, resource discovery module composes a multi-dimensional range query which is the combination of sub-queries on each attribute dimension, and each sub query is responsible to resolve query conditions for a particular attribute in a certain DHT. To perform a range query for a single attribute a_i in the range $[VA(a_i)_{min}, VA(a_i)_{max}]$, resource discovery performs DHT lookup in the range $[H_i(VA(a_i)_{min}), H_i(VA(a_i)_{max})]$. Finally, the discovery results have to satisfy all the single attribute-queries on each attribute dimension. They would be the intersection set of all the individual sub queries results.

Approach C: In the third approach we just select one attribute dimension and establish a DHT for the appointed attributed dimension. The information of the other attribute dimensions is not necessary to be stored in DHT, and it will be stored locally for each resource. To perform range query over multi-dimensional attributes, resource discovery component first performs a range query for the selected single attribute dimension and then the discovery procedure explores the primary set of discovered resources and chooses the ones that meet all the query conditions for the rest of attribute dimensions. In comparison with the previous approach, this method is better in terms of discovery load and traffic generation because it doesn't generate extra sub-queries. However, the distribution of resource information in the previous approach is more balanced.

Approach D: The fourth approach deploys a hierarchical DHT which consists of A hierarchical lookup layers.The discovery request first explores the top layer of the hierarchy that belongs to a certain attribute a_1, which has the maximum priority in comparison with other attribute dimensions. It performs DHT lookup for a single attribute range query in the top layer and then

it narrows the exploring space to a certain set of resources that satisfy the query condition for a_1. In the next level the discovery procedure searches in the second level for a_2, and finally this procedure will end up in last DHT level for a_A. Deploying hierarchical multi-level DHT is similar to the second approach but it creates a significant performance improvement in comparison with the second approach. It eliminates the answers that are not qualified for the single attribute conditions in each level and it makes the exploring space smaller in each level which reduces the discovery traffic load.

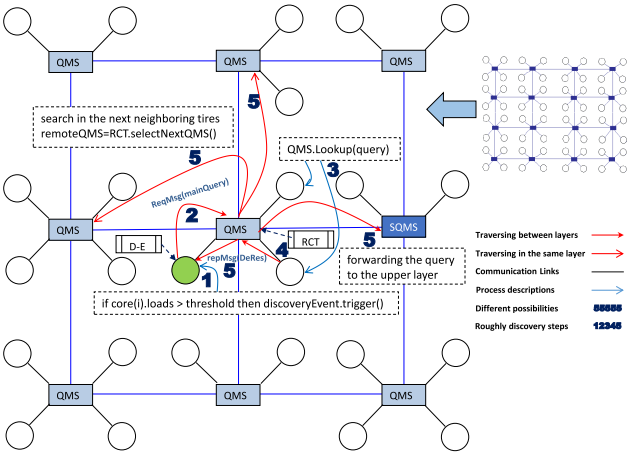

Figure 5: Mechanism of resource discovery according to the cost of resources (The numbers are the sequences of messages)

2.5. *Probability-Based Discovery*

Figure 5 depicts the mechanism of resource discovery in a cluster with multi-core nodes. Finding the optimal resource for a query with minimal requirements depends on processing capability, memory and availability. The search algorithm first explores the possible neighboring nodes and then selects the most promising ones, explores the search graph and goes to the next tier only if the found optimal does not meet the minimal requirements of the query.

We have classified the performance metrics and parameters to evaluate resources in the format of resource description in three individual layers: Core2Core, Die2Die and Node2Node communications level (see Table 1). These parameters handle gathering real time information about the current status of executions and processing capabilities of the available resources. Each group of processors has its own resource cost table (RCT), (see Figure 6), which includes metrics to asses other processors viability to be used as destination of process migration. According to the metrics values in RCT, ranking algorithms generate a rank number for every particular processor in the network. The following figure is a sample of a generic RCT. According to a specific resource description, we must identify the relevant metrics for describing the potential performance of a given resource.

Resource cost tables are maintained in aggregate-nodes and they keep information about the probability of finding a requested resource on different neighbor groups of nodes. They

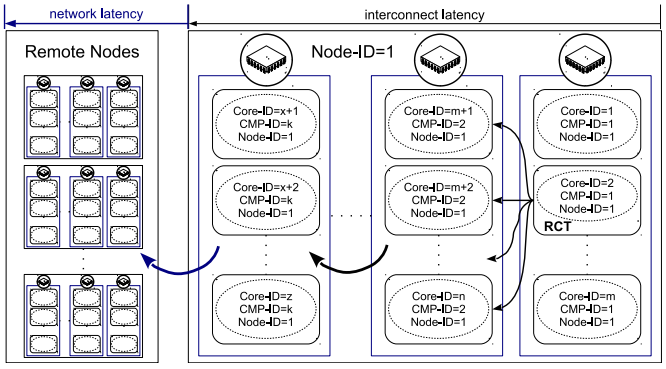

Figure 6: RCT -Assessment of cost of resources per type of query for various ranges of latency

Table 2: Structure of resource cost table for each particular queryTypeID

NID	K2	PF	Probability

aim to evaluate the cost of resources per query type. In order to organize RCT we need to define query types. Table 2 shows the structure of the cost table, where: NID is the neighbor ID; K2 is a binary key that represents the layer2 information; and PF is the probe factor for each neighbor. RCT ranks the neighbor QMSs for each queryTypeID which is conceptually elaborated and based on the range of computation and communication latencies. Before performing query classifications we must estimate optimal or minimal application resource requirements.

Computational clusters are common alternative platforms for handling massively large computational problems and parallel applications. Many-core systems are able to be more efficient when resource management is deployed according to the run-time application requirements. Generating queries for resource discovery requires a capability to predict the resource requirements of parallel applications before making decisions for scheduling. There are a multitude of technologies to estimate and extract the application features which try to deploy performance models or mechanisms to forestall resource utilization in case of computation and communication complexities for applications launched under a distributed parallel system including multiple homogeneous or heterogeneous resources with various processing capabilities[12, 13]. Application modeling or application description can provide accurate information for querying operations in the resource discovery protocol.

At the time of query generation, function of the application features we can assign a queryTypeID for each particular query. After discovering an appropriate resource, all the query-resource mapping information is used to update the probability information and reuse of the discovered resources in similar query statements. For the purpose of simplicity we can assume that we have only one type of query otherwise we have to build a different resource cost table for each query type separately. If we increase the number of resource types it helps the system to be faster and more accurate but it has the drawback of using more storage to keep instances of RCT per queryTypeID.

On – discoveryEvent : query = generateQuery(applicationArguments)

The cost of a resource depends on three aspects, the first one is the processor that is looking for other resources (i.e., the requester) to augment its processing capabilities and distribute jobs among them. It starts the process of discovery and triggers its assigned discovery event. The second element is the application that runs on the aforementioned processor. There are a variety of applications with different specifications and requirements but for simplicity we assume that for each particular application we can map it to a queryTypeID, where the type of query describes the application execution's resource requirements. The final element is the processor that would be nominated as the result of the discovery. Considering these elements and their effects on the resource evaluation, we can model the complexity of the assessment as follows:

$ResourceCost = CommunicationTime + ComputationTime$

In classic resource discovery protocols, a typical cluster environment is generally defined as a Virtual Organization (VO) which consists of two types of individuals: resources and resource directories. In our approach, for discovery we assume that a cluster (i.e., The QMS region) is a directed graph $Cg(Ve, Ed)$ with n nodes and w edges. Each edge $ed(a, b)$ connects to two nodes a and b where:

$ed(a, b) \in Ed \wedge \forall a, b \in Ve \mid a, b \leq n$

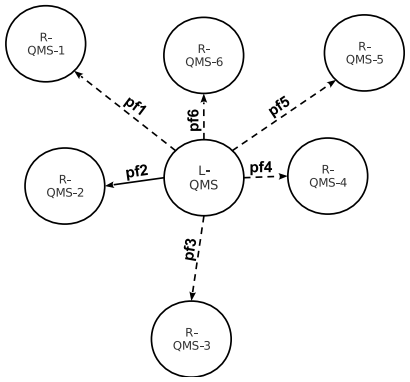

Figure 7: Selection of the consequence QMS based on values of probe factors and probabilities

We assign a probe factor Pf to each one of the links from local QMS to the neighboring QMSs. Nodes in the graph are cores in real cluster. Cores are linked to each other through interconnection networks. We assume that all the cores which are placed in a CPU or cluster node are members of a QMS. $ed(a, b)$ is the connection link between core a and core b, where communication cost to describe it are bandwidth and delay. For each core, states and descriptions of its processing resource and also other group of cores in the first-tier vicinity are stored in localQMS. Cost of resource r in localQMS of core b through $ed(a, b)$ maintained as $Rc^r_{ab} = \frac{1}{Pf^r_{ab}}$ which Pf is the probe factor for the existence of resource r with resource type of t in the QMS of b when the query is going to be forwarded from the QMS of a (see Figure 7).

In our resource discovery approach, we generate one query per discovery request, and in the whole cluster system according to the number of requesters, several queries would be generated concurrently. They explore different paths in line with the link parameters like probe factor. However, the values of the probe factors are updated by all those query replies that have completed a discovery procedure.

Queries explore the cluster and check local and remoteQMS for the best resource matching. In fact, when a query explores a QMS and related resources, it shows one of the following two behaviors: First, query does not find the requested resource, then it goes to the next QMS, and continues until the best matching for discovery is achieved or the query is expired. If a query that is exploring QMS associated with aggregate-node a is not satisfied with the local resources it goes to the next aggregate-node b in one of the QMS neighbor with the following probability which is proportionally related to the rate of Pf^r_{ab} for each one of r resources over edge $ed(a, b)$:

$Probability^r_{ab} = \frac{Pf^r_{ab}}{\sum_z Pf^r_{az}}$

z is an aggregate-node which is in vicinity of a. And r is a resource type which probably existed in the QMS of b. According to the above formula, queries use the probability to select the next QMS for discovery. After finding a match for a query, the reply message includes the description of the qualified discovered resources which will be returned to the initial requester. Among all the potential resources the best matching is a resource with lowest rate of latency. In the next step the probe factor for the result of the performed discovery procedure would be updated according to following formula:

$$Pf^r_{ab} = \begin{cases} Pf^r_{ab} - \delta Pf^r_{ab} + \frac{\delta}{L_{ab}} & if\ r \in QMS_b \\ \delta Pf^r_{ab} & if\ r \notin QMS_b \end{cases}$$

δ is a random number which is $0 < \delta \leq 1$ and L_{ab} is the latency between a and b. r is the requested resource with type t. The default value of Pf is 1.

3. Evaluation

In this section we describe simulation results regarding the resource discovery evaluation in multi/many-core networked environments. A key factor in our evaluation is showing the functionality and scalability of the RD system. To achieve this we have set up a virtual networked environment with a number of multi-core / multi-processor nodes where all the processors are enabled to invoke the resource discovery module in order to find other possible alternative resources for the transaction of the overloaded processes to remote processors. The simulation experiments have been conducted based on the COTSon Simulator[14] and we simulated a networked environment with 23 multi-core nodes (each node has 2, 4, 6 or 8 cores) including the total number of 88 resources (cores). Scalability for RD means the ability of the discovery mechanism to support a larger number of resources or a greater number of inter-operations between the nodes, or both. We analyze the RD scalability by measuring discovery latency versus the number of nodes where one, or a constant fraction of the nodes, or a percentage of the total nodes, concurrently and periodically (with different intervals) generate a query and trigger the resource discovery module to get information about other nodes in a congested network. We analyzed the impact of the network size on the discovery

delay which is the response time to get information. To achieve scalability, discovery delay should not have a big variation when we change the size of network.

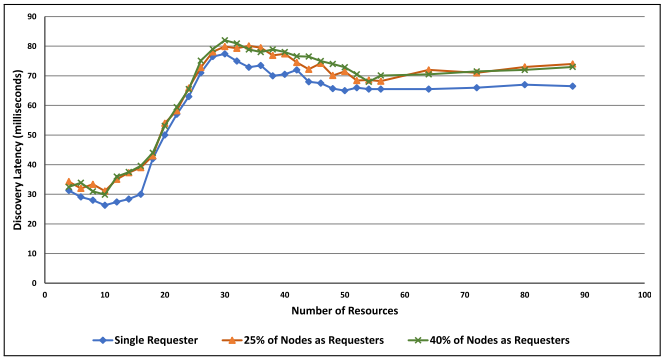

Figure 8: Average Discovery Latency for Single Requester, 25% and 40% of nodes as requesters with interval 30s

We evaluated the impact of the network size on the resource discovery latency for different number of requesters. After a random initialization time, each requester starts submitting a query to the system every 30s. We have run 1000 RD queries for each requester in the given network size. The minimum query dimension is 4 and the maximum number of attributes is 20. We also vary the number of desired resources for each resource discovery query from 1 to 4. The properties of each query and also the network topology are random. Accordingly, dependent on the network topology and network size in each run time, the simulation scenarios with different number of resource information providers (i.e., the nodes with QMS support) as distributed directories for our simulated network, would be established. However, the number of resource providers would be fixed during the simulation run time.

In Figure 8, while we increase the number of resources, the discovery response time starts to increase but for large number of resources it remains almost constant. In all these experiments that were done with different number of requesters we roughly see the same behavior in the results. This shows that discovery latency is independent from network size which means that for large scale networks we could still have a reasonable response time for resource discovery queries.

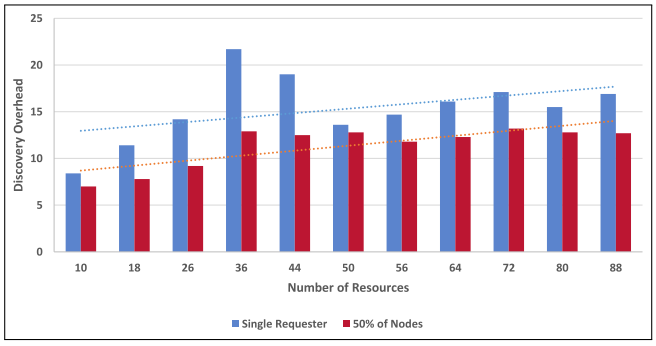

Figure 9: Average Discovery Overhead in different network size for 1 requester and 50 % of nodes as requesters with Query Rate=.033/s

In the second test we measured the average of the discovery overhead versus the number of nodes. This test is to analyze the impact of network size on the average number of transaction messages for each discovery process. Figure 9 plots the discovery overhead of our system network sizes in the two configurations. In the first configuration only one of the nodes generates a random query every 30 seconds. By increasing the network size first, the overhead (the average number of discovery messages per query) increases until it gets to a threshold and then it decreases slightly and remains almost constant for large network sizes. By augmenting the network size the RD system is required to forward the queries to other remote directories with larger distance in the system to find the resources around the network. Therefore, increasing the network size leads to increasing the search radius which would eventuate in propagating more messages for resolving the discovery requests by efficiently exploring the discovery region. Thus we could expect to see the growing of the overhead with the increasing of the number of resources.

The overhead starts to reduce after reaching the maximum due to the increasing number of remote directories that provide information about other alternative resources and also the increase of the replication of the resources. Interestingly, in a second configuration that we have simulated for a congested network with 50 percent of the nodes as concurrent requesters, the results show a significant decrease in the discovery overhead. The reason for this overhead reduction is that when we perform several queries concurrently by different distributed RD components (i.e., resource requesters), the probability tables of intermediate directories (QMS nodes) would dynamically be updated according to each discovery result. This will improve the degree of resource awareness in the probability tables in the directories which would lead to the reduction of the discovery overhead (i.e., reducing the number of forwarding and query dissemination) compared to the first configuration. Moreover, implementation of the resource caching mechanism in the nodes of different types (either LN, AN or SN nodes) enables the general nodes to store the experience of successful queries for a specific period of time which facilitates the resolving of similar queries from other requesters with less overhead cost. In both configurations the overhead variations are reasonably small for large network sizes. Therefore the overhead should not be dependent on the network size which means that our RD solution is scalable.

4. Related Work

There are many resource discovery models that have been developed in the area of large scale distributed computing, but few were proposed to address the complexities of many-core, and even less are able to do so efficiently. Distributed RD proposals have mostly been designed to provide a high level of scalability and fault tolerance, which is required in large scale environments. Iamnitchi et al [15] proposes a solution for using the benefit of distributed Peer-to-Peer systems for resource discovery in Grids. The combination of P2P and Grid RD models [16] would be desirable to build fault tolerant and large scale distributed systems. There are two kind of approaches in this field which are based on

structured and unstructured overlays networks. The first model uses a unstructured overlay network with flooding based query propagation. Relevant solutions are Zorilla [17] and Vishwa[18]. One of the advantages of this approach is the ability to perform resource discovery with high expressiveness. However, such discovery systems are not exhaustive and efficient. The response time of the queries is high due to flooding and blind search. Also, rare resource information may be unable to be found.

There also exist some operating systems and cluster management systems which are using their own designed resource discovery components such as Plan9 [19] and OpenMOSIX [20].

All of these techniques suffer from at least one of the following problems: i) low scalability (e.g.,Condor [21], Condor-G [22], BOINC [23] and etc.,), ii) low efficiency for structured environments(e.g., Globus [24], Routing Indices (RI) [25], etc.,), iii) inadequate resource descriptions and matching for many-core environments where hardware devices are the resources to be propagated (e.g., NodeWiz [26] , Mercury[27] and SWORD[28]).

5. Conclusions

In a large scale system, where we have a pool of distinct processors, the enabling technology for enhancing the whole throughput of the system is resource sharing. This means that for overloaded processors we can migrate the overloaded processes to other potential processors in the network. But resources should be found before the resource sharing, resource allocation and execution migration could be done. Resource discovery as a component of a distributed OS will be employed to discover an efficient set of available processing resources that could match the application requirements. The discovery latency has direct effect on the cost of migration and execution migration is not beneficial when resource discovery cannot provide information services in an acceptable time. Unlike resource discovery for various other purposes and domains, resource discovery for a large many-core environment is very sensitive to the discovery performance and it could be useless when it cannot satisfy the minimal parametric conditions of the system environment.

Scalability could be considered as one of the basic problems of resources discovery in future many-core systems which is a generic challenge for majority of the research works in this area. The scalability problems also refer to the methods for description of resources and the discovery procedure. These mechanisms must propose techniques and algorithms that are efficiently extend-able for various number of resources. On the other hand the generated discovery overhead must be independent from network size. This is not fully attainable but we can make an effort to keep the discovery overhead almost constant with increasing the number of resources.

In this paper we have indeed presented a new resource discovery solution which is scalable and efficient for future many-core systems with a huge number of resources. We established a distributed system with hierarchical resource description and designed a probability-based Any-cast mechanism to locate resources in hierarchical levels of core-chip-node-network. Simulation results show that the proposed RD supports efficiency and scalability for discovery in large distributed systems.

6. Acknowledgment

The authors acknowledge the support of project FP7-ICT-2009.8.1, Grant Agreement No.248465, Service-oriented Operating Systems (2010-2013)[4, 5] and of project Cloud Thinking (2013-2015), CENTRO-07-ST24-FEDER-002031[29].

References

[1] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software, Dr. Dobb's Journal 30 (2005).

[2] A. Clematis, A. Corana, A. Galizia, A. Quarati, Matching jobs with resources: An application-driven approach, SCPE: Scalable Computing, Practice and ExperienceScientific, International Journal for Parallel and Distributed Computing 11 (2010).

[3] A. Tikotekar, G. Vallée, T. Naughton, H. Ong, C. Engelmann, S. L. Scott, Euro-par 2008 workshops - parallel processing, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 63–71.

[4] L. Schubert, A. Kipp, Principles of service oriented operating systems, in: P. Vicat-Blanc Primet, T. Kudoh, J. Mambretti (Eds.), Networks for Grid Applications, volume 2 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer Berlin Heidelberg, 2009, pp. 56–69.

[5] The S[o]OS Consortium , S(o)OS (Service-oriented Operating System): Resource-independent execution support on exa-scale systems, `http://www.soos-project.eu/`, 2010-2013. [Online: accessed 5-September-2014].

[6] E. Meshkova, J. Riihijärvi, M. Petrova, P. Mähönen, A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks, Computer Networks 52 (2008) 2097–2128.

[7] Y.-C. Wu, C.-M. Liu, J.-H. Wang, Enhancing the performance of locating data in chord-based p2p systems, in: Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on, pp. 841–846.

[8] B. Wiley, Distributed hash tables, part i, Linux J. 2003 (2003) 7–.

[9] W. Gropp, E. Lusk, A taxonomy of programming models for symmetric multiprocessors and smp clusters, in: Programming Models for Massively Parallel Computers, 1995, pp. 2–7.

[10] S. Zoels, Z. Despotovic, W. Kellerer, On hierarchical dht systems - an analytical approach for optimal designs, Comput. Commun. 31 (2008) 576–590.

[11] J. R. Flynn, The flynn effect and flynns paradox, Intelligence 41 (2013) 851–857. ¡ce:title¿The Flynn Effect Re-Evaluated¡/ce:title¿.

[12] J. Kuntraruk, Application Resource Requirement Estimation in a Parallel-pipeline Model of Execution on a Computational Grid, Ph.D. thesis, Bethlehem, PA, USA, 2003. AAI3117162.

[13] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, T. von Eicken, Logp: A practical model of parallel computation, Commun. ACM 39 (1996) 78–85.

[14] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, D. Ortega, Cotson: Infrastructure for full system simulation, SIGOPS Oper. Syst. Rev. 43 (2009) 52–61.

[15] Iamnitchi, I. T. Foster, On fully decentralized resource discovery in grid environments, in: the Second International Workshop on Grid Computing (Grid'01), Springer-Verlag, London, UK, 2001, pp. 51–62.

[16] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, S. Haridi, Peer-to-peer resource discovery in grids: Models and systems, Future Generation Computer Systems 23 (2007) 864–878.

[17] N. Drost, et al., Zorilla: a peer-to-peer middleware for real-world distributed systems, Concurrency and Computation-Practice & Experience 23(13) (2011) 1506–1521.

[18] M. Reddy, et al., Vishwa: A reconfigurable p2p middleware for grid computations, in: International Conference on Parallel Processing, pp. 381–388.

[19] A. Mirtchovski, R. Simmonds, Plan 9 - an integrated approach to grid computing, in: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 273.

[20] R. Ho, C.-L. Wang, F. Lau, Lightweight process migration and memory prefetching in openmosix, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, pp. 1–12.

[21] T. Tannenbaum, M. Litzkow, The condor distributed-processing system, Dr Dobbs Journal 20(2) (1995) 40–&.
[22] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, Condor-g: A computation management agent for multi-institutional grids, Cluster Computing 5 (2002) 237–246.
[23] D. Anderson, Boinc: A system for public-resource computing and storage, in: Fifth Ieee/Acm International Workshop on Grid Computing, pp. 4–10.
[24] X. Zhang, J. Schopf, Performance analysis of the globus toolkit monitoring and discovery service, mds2, in: the 2004 Ieee International Performance, Computing, and Communications Conference, pp. 843–849.
[25] A. Crespo, H. Garcia-Molina, Routing indices for peer-to-peer systems, in: Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on, pp. 23–32.
[26] S. Basu, et al., Nodewiz: Peer-to-peer resource discovery for grids, in: 2005 IEEE International Symposium on Cluster Computing and the Grid, volume 1 and 2, pp. 213–220.
[27] R. Devarakonda, et al., Mercury: reusable metadata management, data discovery and access system, Earth Science Informatics 3(1-2) (2010) 87–94.
[28] J. Albrecht, et al., Design and implementation trade-offs for wide-area resource discovery, Acm Transactions on Internet Technology 8(4) (2008).
[29] R. Aguiar, D. Gomes, J. Barraca, N. Lau, Cloudthinking as an intelligent infrastructure for mobile robotics, Wireless Personal Communications 76 (2014) 231–244.